[SYSTEM-DESIGN]

Designing Data-Intensive Applications – Chapter 4: Encoding and Evolution

Posted by CHARLES *on* 2020-04-06

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

Everything changes and nothing stands still. —Heraclitus of Ephesus, as quoted by Plato in Cratylus (360 BCE)

- Applications inevitably change over time..
- schema-on-read ("**schemaless**") databases don't enforce a schema, so the database can contain a mixture of older and newer data formats written at different times
- When data changes usually you will need code change, but in a large application, code changes often cannot happen instantaneously.
  - With server-side applications you may want to perform a **rolling upgrade** (also known as a **staged rollout**)
  - With client-side applications you're at the mercy of the user, who may not install the update for some time.
- We need to maintain compatibility in both directions
  - **Backward compatibility**: Newer code can read data that was written by older code. (relative easy, since you already knew the existing old data schema)
  - **Forward compatibility**: Older code can read data that was written by newer code. (relative harder, it requires older code to ignore additions made by a newer version of the code)

Formats for Encoding Data:

- The translation from the in-memory representation to a byte sequence is called **encoding** (also known as **serialization** or **marshalling**), and the reverse is called **decoding** (**parsing**, **deserialization**, **unmarshalling**).
- **Language-Specific Formats**: (bad idea, usually don't recommended)

    - The encoding is often tied to a particular programming language, and reading the data in another language is very difficult.
    - In order to restore data in the same object types, the decoding process needs to be able to instantiate arbitrary classes.(Potential security risk)
    - **Versioning** data is often an afterthought in these libraries
    - **Efficiency** (CPU time taken to encode or decode, and the size of the encoded structure) is also often an afterthought
- **JSON, XML, and Binary Variants**:
  - Problems:
    - There is a lot of ambiguity around the encoding of numbers. This is a problem when dealing with large numbers(e.g. 2^53);
    - JSON and XML have good support for Unicode character strings (i.e., human- readable text), but they don't support binary strings
    - There is optional schema support for both XML and JSON (but complicated to learn and use)
    - CSV does not have any schema, so it is up to the application to define the meaning of each row and column.
  - Despite these flaws, JSON, XML, and CSV are good enough for many purposes.
  - **The difficulty of getting different organizations to agree on anything outweighs most other concerns**;
- **Binary encoding**: For data that is used only internally within your organization, there is less pressure to use a lowest-common-denominator encoding format
  - JSON is verbose but it use lots of space, hence it led to the development of a profusion of binary encodings for JSON (MessagePack, BSON, BJSON, UBJSON, BISON, and Smile, to name a few) and for XML (WBXML and Fast Infoset, for example) [but they wasn't save too much space neither]
- **Thrift and Protocol Buffers**: Apache Thrift [by Facebook] and Protocol Buffers (protobuf) [by Google] are binary encoding libraries that are based on the same principle.
  - Both Thrift and Protocol Buffers
    - require a schema for any data that is encoded.
    - each come with a code generation tool
  - **Thrift** has two different binary encoding formats,
    - BinaryProtocol
    - CompactProtocol (and DenseProtocol only support C++)
  - **Protocol** only have Binary encoding;
    - does not have a list or array data type, but instead has a repeated marker for fields (which is a third option alongside required and optional).
  - **Field tags and schema evolution**:
    - to maintain backward compatibility, every field you add after the initial deployment of the schema must be optional or have a default value.
    - you can only remove a field that is optional (a required field can never be removed), and you can never use the same tag number again

    - **The writer's schema and the reader's schema**:
        - The key idea with Avro is that the writer's schema and the reader's schema don't have to be the same—they only need to be compatible.
    - **Schema evolution rules**:
        - To maintain compatibility, you may only add or remove a field that has a default value.
    - In Avro:
        - if you want to allow a field to be null, you have to use a union type. For example, union { null, long, string } field
        - Doesn't have optional and required markers.
        - Change field name: the reader's schema can contain aliases for field names, so it can match an old writer's schema field names against the aliases.
    - A database of schema versions is a useful thing to have in any case, since it acts as documentation and gives you a chance to check schema compatibility.
    - **Dynamically generated schemas**
    - Code generation and dynamically typed languages:
        - After a schema has been defined, you can generate code that implements this schema in a programming language of your choice.
- **The Merits of Schemas:**
    - schema evolution allows the same kind of flexibility as schemaless/ schema-on-read JSON databases provide, while also providing better guarantees about your data and better tooling.

Modes of Dataflow

- Forward and backward compatibility, which are important for **evolvability** (upgrade system independently, and not having to change everything at once)
- **Via Databases**:
    - **Different values written at different times**: A database generally allows any value to be updated at any time. (aka, **data outlives code**) using Schema evolution rules.
    - **Archival storage**: use latest schema
        - encode the data in an analytics-friendly column-oriented format such as Parquet
- **Via Synchronized Service call**: **REST** or **RPC**
    - **REST** seems to be the predominant style for **public** APIs.
    - **RPC** frameworks are on requests between services owned by the same organization, typically within the same datacenter.
    - The most common arrangement is to have two roles: **clients** and **servers**.
        - The API exposed by the server is known as a **service**.
    - Moreover, a server can itself be a client to another service (e.g. a web app server acts as client to a database)
        - **service-oriented architecture (SOA)**, more recently refined and rebranded as **microservices architecture**;

- **Web services**: **REST** and **SOAP**
  - **REST** is not a protocol, but rather a **design philosophy** that builds upon the principles of HTTP. An API designed according to the principles of REST is called RESTful.
    - A definition format such as **OpenAPI**, also known as **Swagger**, can be used to describe RESTful APIs and produce documentation.
  - **SOAP** is an XML-based protocol for making network API requests.
    - The API of a SOAP web service is described using an XML-based language called the **Web Services Description Language, or WSDL**, WSDL enables code generation so that a client can access a remote service using local classes and method calls; (good for static typed language like Java)
- **RPC**:
  - The RPC model tries to make a request to a remote network service look the same as calling a function or method in your programming language, within the same process (this abstraction is called **location transparency**).
  - Although RPC seems convenient at first, the approach is **fundamentally flawed**. A network request is very different from a local function call.
    - A network request is unpredictable;
    - A network response is unpredictable;
    - Idempotence is required for re-try;
    - A network request is much slower than a function call, and its latency is also wildly variable;
  - **Current directions for RPC**: Thrift and Avro come with RPC support included, gRPC is an RPC implementation using Protocol Buffers, Finagle also uses Thrift, and Rest.li uses JSON over HTTP.
    - Custom RPC protocols with a binary encoding format can achieve better performance than something generic like JSON over REST;
    - But REST it is good for experimentation and debugging;
  - Data encoding and evolution for RPC

- **Via Asynchronous message passing**: asynchronous message-passing systems, which are somewhere between RPC and databases.
  - **Advantages**:
    - Act as buffer;
    - Auto-redeliver message;
    - Allow message fan-out;
    - Decouple the sender and recipient;
  - **Disadvantages**:
    - message-passing communication is usually one-way: a sender normally doesn't expect to receive a reply to its messages
    - communication pattern is asynchronous: the sender doesn't wait for the message to be delivered, but simply sends it and then forgets about it

ensures that the message is delivered to one or more **consumers** of or **subscribers** to that **queue** or **topic**.
- There can be many **producers** and many **consumers** on the same topic.
- A topic provides only one-way dataflow. However, a consumer may itself publish messages to another topic (so you can chain them together), or to a reply queue that is consumed by the sender of the original message (allowing a request/response dataflow, similar to RPC)
- **Distributed actor frameworks**: The actor model is a programming model for concurrency in a single process. (e.g. Akka, Orleans, Erlang OTP)
  - A distributed actor framework essentially integrates a message broker and the actor programming model into a single framework.

Summary:

- many services need to support rolling upgrades → evolvability
  - released without downtime
  - make deployments less risky
- all data flowing around the system is encoded in a way that provides backward compatibility (new code can read old data) and forward compatibility (old code can read new data).
  - data encoding formats and it pros and cons;
- several modes of dataflow, illustrating different scenarios in which data encodings are important.
  - **Databases**, where the process writing to the database encodes the data and the process reading from the database decodes it.
  - **RPC and REST APIs**, where the client encodes a request, the server decodes the request and encodes a response, and the client finally decodes the response
  - **Asynchronous message passing** (using message brokers or actors), where nodes communicate by sending each other messages that are encoded by the sender and decoded by the recipient

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

PREVIOUS POST
Designing Data-Intensive Applications – Chapter 2: Data Models and Query Languages

NEXT POST
Designing Data-Intensive Applications – Chapter 5: Replication